

# C++20 per impazienti





# Italian C++ Conference 2019

Thank you! ❤



# Why not enabling C++20?

Some real answers I got:

«Too many things to study. Need more time to get up to date and train the team»

«I don't need Concepts, Ranges, Modules and Coroutines»

«They have messed up with direct initialization. Can I ask for a refund?»

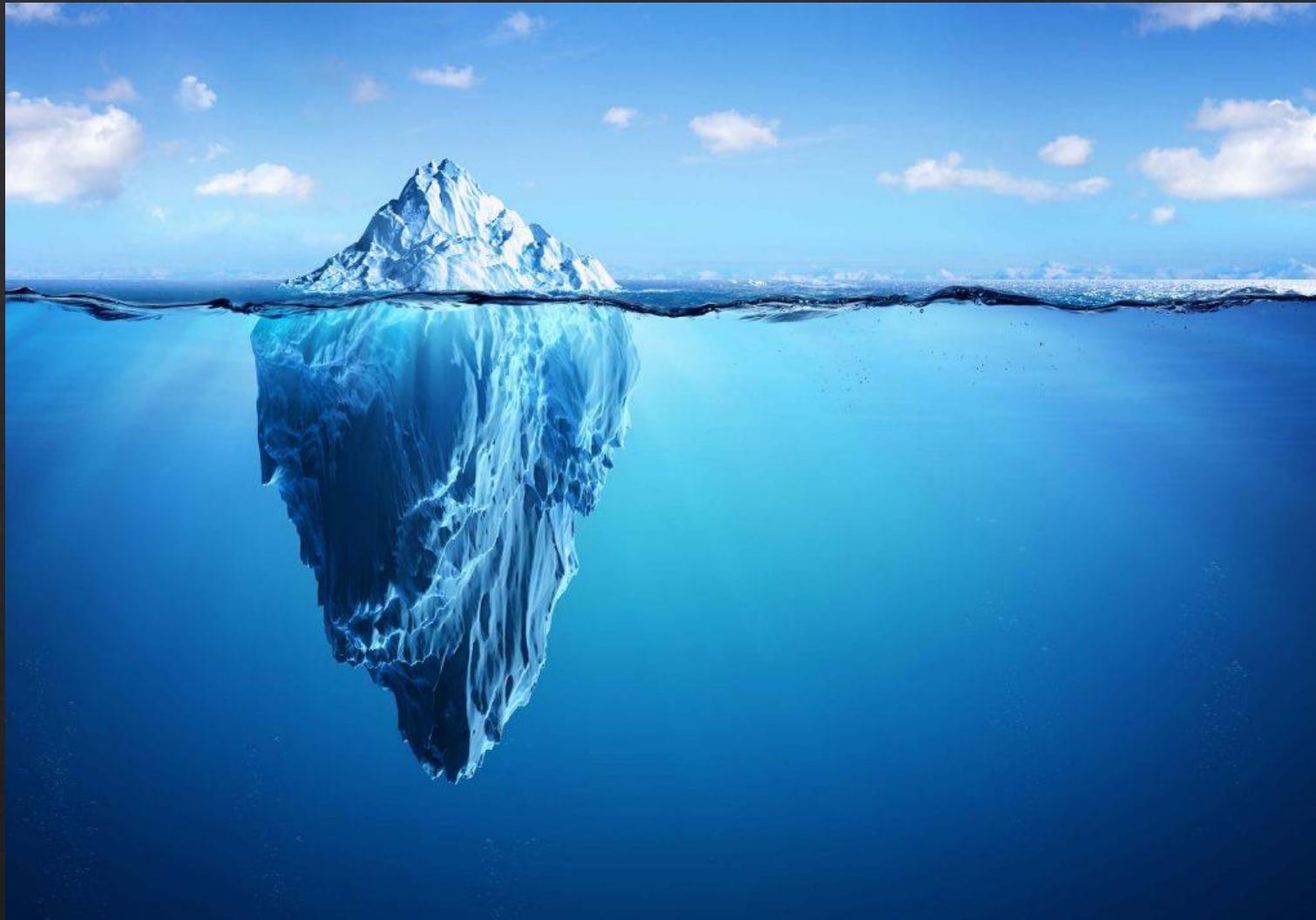
«It's not available on my compiler»



Lasciatevi ispirare.



Let yourself inspire



Beware of complexity

# Comparisons

# 3-way comparison "spaceship" operator

- ❖ TL;DR: we can get all comparison operators automatically generated
- ❖ By declaring

`operator<= = default,`

the compiler automatically generates a corresponding equal operator

`operator== = default`

```
struct MyType
{
    auto operator<= (const MyType&) const = default;
    auto operator== (const MyType&) const = default; // implicitly generated
};
```

# 3-way comparison "spaceship" operator

```
struct Point3D
{
    auto operator<=> (const Point3D& other) const = default;
    int x, y, z;
};

Point3D p1 {1, 0, 0}, p2 {1, 0, 1};
cout << (p1 == p2); // 0
```

# 3-way comparison "spaceship" operator

That's it...?



Well, no...  
...this is C++, after all

# 3-way comparison "spaceship" operator

- ❖ `operator== = default` automatically performs member-wise comparison
- ❖ `operator== implies operator!=`

TL;DR:  $a \neq b \rightarrow$  the compiler tries to find  $a \neq b$ ,  $!(a == b)$ , and  $!(b == a)$

The compiler is actually able to automatically **rewrite** the expression

- ❖ Introduced `operator<=` that automatically **synthesizes** all the relational operators (memberwise) by performing **rewriting**. For example the compiler:

to evaluate  $x \leq y$ , first tries to find viable `operator<=`, otherwise

$$(x \leq y) \leq 0$$

or

$$0 \leq (y \leq x)$$

# 3-way comparison "spaceship" operator

The result of `operator<=:`:

- ❖ **is comparable with 0** – classical meaning from C, e.g.  $(x \ltreq y) < 0$  means that  $x < y$
- ❖ its **type** represents the **comparison category** (**strong**, **weak**, **partial ordering**)
- ❖ Stronger comparison categories are implicitly convertible to weaker ones
- ❖ Formally included in `<compare>`



<https://wandbox.org/permlink/uDWEI3aYH7whzyuo>

# Strong, weak, partial...oh my!

Category	Sostituitability	Equivalence	Allows incomparable values
strong_ordering	Yes	Same as equality	No
weak_ordering	No	$!(a < b) \ \&\& \ !(b < a)$	No
partial_ordering	No	$!(a < b) \ \&\& \ !(b < a)$	Yes (unordered)

# cmp\_\* functions

```
-1 < 0u; // false :(  
std::cmp_less(-1, 0u); // true :)
```



<https://wandbox.org/permlink/YwM3C51paiPQH3XC>

# Initialization

# Designated initializers

```
struct Person
{
    std::string name;
    int code;
    bool flag = false;
};
```

```
Person p { .name="marco", .code=10, .flag=true};
```

# Designated initializers

- ❖ Only for **aggregate** types
- ❖ Compatible with **C**, however in **C++** we **do not** support:
  - ❖ out of order
  - ❖ nesting (we have to do it explicitly)
  - ❖ arrays
  - ❖ mix with regular initializers
- ❖ Does not work with **inheritance** (we have a proposal)



<https://wandbox.org/permlink/nOvqAHZif0PV4yXU>

# Designated initializers

```
struct Person
{
    std::string name;
    int code;
    bool flag = false;
};
```

```
Person p {.name="marco", .code=10, .flag=true}; // ok
Person p {.name="marco", .flag=true}; // ok, code is 0
Person p {.name="marco", .flag=true, .code=10}; // nope (out of order)
Person p {.name="marco", .code=10.6, .flag=true}; // nope (narrowing)
Person p {.name="marco", .code=10.6, true}; // nope (not everywhere)
```

# Designated initializers

```
struct Point
{
    int x;
    int y;
};

struct Line
{
    Point p1;
    Point p2;
};

// nesting now allowed. Workaround:
Line l { .p1={ .x=10, .y=20 }, .p2={ .x=50, .y=60 } };
```

# Aggregate initialization with ( )

( ) is now able to perform **aggregate initialization**:

```
struct Person
{
    string name;
    int code;
    bool flag = true;
};

Person p{"marco", 15};
```

```
auto p1 = make_unique<person>("marco", 15); // ok in C++20
```

```
auto p2 = make_unique<person>("marco", 15, false); // ok in C++20
```

# Aggregate initialization with ( )

- ❖ ( ) is now able to perform **aggregate initialization**...however:
  - ☒ does not call `std::initializer_list` constructors
  - ☒ does not work with designated initializers
  - ☒ does not allow brace elision
  - ☒ can't extend lifetime of a temporary bound to a reference



<https://wandbox.org/permlink/67b4worFFR2go5ED>

# Range-based for loop with initializers

```
for (const auto& entry = getCache().Values())
{
    entry.do_something(); // UB if getCache() returns by value
}

// now we can do:

for (auto cache = getCache(); const auto& entry = cache.Values())
{
    entry.do_something();
}
```

# Using enums

```
enum class LogLevel {  
    trace, debug, info, warning, error, fatal  
};  
std::string_view ToString(LogLevel level) {  
    switch(level) {  
        using enum LogLevel;  
        case trace:    return "trace";  
        case debug:   return "debug";  
        case info:    return "info";  
        case warning: return "warning";  
        case error:   return "error";  
        case fatal:   return "fatal";  
    }  
}
```



<https://wandbox.org/permlink/OiNsXjv8nIb0pL6D>

# Lambda Additions

# Lambda-capture refinements

```
// 1. structure binding identifiers in capture list
auto [name, value] = cache.GetEntry("key");
auto printer = [name](int i) {
    return name + to_string(i);
};

// 2. pack expansions in init-captures
template<typename... Args>
void capture_init_this(Args... args) {
    auto f = [...args=std::move(args)]() { ... };
}
```



<https://wandbox.org/permlink/tzzqOYtUuSSouGhB>

# Lambdas usability changes

- ❖ **Stateless lambdas** = lambdas with no capture
  - ❖ They now provide a defaulted constructor (=default)
  - ❖ They can now be copy-assigned and move-assigned

```
auto ciCompare = [](string_view left, string_view right) {  
    const auto cmp = strncasecmp(left.data(), right.data(), std::min(left.size(), right.size()));  
    return cmp == 0 ? (left.size() < right.size()) : cmp < 0;  
};  
  
set<string, decltype(ciCompare)> names;  
// until C++20 we needed to construct the callable object explicitly:  
// set<string, decltype(ciCompare)> names { ciCompare };
```

- ❖ Lambdas can now be used in **unevaluated contexts** (e.g. `sizeof`, `decltype`)



<https://wandbox.org/permlink/uaruK1EwagecIFzj>

# Template syntax for generic lambdas

- ❖ We can now have **explicit names for generic parameters in a lambda**
- ❖ Useful for **restricting** the type of parameters of a lambda (as an alternative to **type constraints**)

```
auto bar = []<typename T>(const T & param) {  
    T tmp{};  
    // ...  
};  
  
// for example, this will be called on vectors only  
auto onlyVec = []<typename T>(const std::vector<T>& vec) {  
    //...  
};
```

# Templates

# auto syntax for generic functions

- ❖ Syntactic sugar to declare/define template functions
- ❖ All rules of using function templates apply

```
void foo(const auto& bar);
```

// equivalent to:

```
template<typename T>
void foo(const T& bar);
```

Lambdas get close to functions...  
...functions get close to lambdas



# NTTP - Non-Type Template Parameters

- ❖ We can now use more types as template parameters:
  - ❖ Floating point types
  - ❖ Simple classes - formally called **structural types** (literal types with all public members)
  - ❖ Lambdas (`constexpr` – which are, when possible, by default since C++17)

```
struct Discount
{
    double value;
};

template<Discount D>
auto CalculateTotalPrice(int price)
{
    return price * D.value;
}

cout << CalculateTotalPrice<Discount{0.9}>(35);
```

```
template<double D>
auto CalculateTotalPrice(int price)
{
    return price * D;
}

cout << CalculateTotalPrice<0.7>(35);
```

```
template<auto D>
auto CalculateTotalPriceLambda(int price)
{
    return price * D();
}

auto lambdaDiscount = []{
    return 0.5;
};

cout << CalculateTotalPriceLambda<lambdaDiscount>(35);
```



<https://wandbox.org/permlink/urfYpgwh0EiavGMx>

# CTAD - Class Template Argument Deduction

- ❖ Some additions have been made to support:
  - ❖ Aggregate types
  - ❖ Alias templates
  - ❖ Non-type template parameters

```
template<typename T1, typename T2>
struct my_pair {
    T1 t1;
    T2 t2;
};

my_pair p = {10, true};

// until C++20 we needed an explicit deduction guide:
// template<typename T1, typename T2> my_pair(T1 t1, T2 t2) -> my_pair<T1, T2>;
```

# Threads & Concurrency

# std::osyncstream

- ❖ We can now **synchronize** the concurrent output of multiple threads to the same stream
- ❖ Each thread **must** use its own **osyncstream** (concurrent writes to the same stream is UB)
- ❖ The output is written only when the destructor of **osyncstream** is called
- ❖ The manipulator **flush\_emit** has been introduced to bring the writing forward

```
std::osyncstream wrapped(std::cout);  
  
wrapped << "hello osyncstream\n";
```



<https://wandbox.org/permlink/BrdgB4h4eJx1XxhE>

# std::jthread

- ❖ From a resource point of view, `std::thread` is just like an owning raw pointer
- ❖ `jthread` is a new *RAII type* whose destructor calls `join()`, if the thread is `joinable`
- ❖ It's not a wrapper on top of `std::thread`

```
int main()
{
    jthread t { []{
        cout << "hello jthread";
    } };
    cout << "hello main";
} // t.join() called automatically
```

# std::jthread

- ❖ Provides a **cooperative** mechanism to signal and query **cancellation**
- ❖ std::jthread's destructor signals cancellation **and then** calls join(), if the thread is **joinable**
- ❖ Also, can be explicitly requested by calling **request\_stop()**

```
jthread t { [](stop_token st) {
    while (!st.stop_requested()) {
        // ...
    }
} };
...
t.request_stop();
```

# Cooperative cancellation

- ❖ The cancellation mechanism introduced for jthread is actually more generic
- ❖ C++20 introduces **stop sources** and **tokens** to **synchronize a shared stop state**
- ❖ `std::stop_source` is used to **request a stop**
- ❖ `std::stop_token` is used to **react to a stop** (via polling or `stop_callback`)
- ❖ Sources and tokens can be copied and moved around (with expected semantics)
- ❖ `std::stop_token` can be passed to the waiting functions of `std::condition_variable_any`

```
stop_source src; // creates a new shared stop state
auto token = src.get_token();
start_task(fun, token); // imagine to "fire and forget" fun
// ...
src.request_stop(); // this request is synchronized
```

# std::jthread and cancellation

- ❖ jthread has its own stop\_source
- ❖ You can pass your own stop\_token

```
stop_source src; // creates a new shared stop state  
  
auto token = src.get_token();  
  
jthread t1 {[](stop_token st) { ... }, token);  
jthread t2 {[](stop_token st) { ... }, token);  
  
// ...  
  
src.request_stop(); // request both t1 and t2 to stop
```

# std::stop\_callback

- ❖ Another way to react to a stop request is by registering one or more `std::stop_callback`
- ❖ Called automatically when a stop is requested (deregistered automatically if the thread finishes before)
- ❖ The callable is executed by the **thread requesting the stop\***
- ❖ **Order** in which more callbacks will be executed is **not guaranteed** (but they will be invoked **synchronously**)
- ❖ Callables cannot exit via an exception (otherwise `std::terminate` is called)

```
jthread t { [](stop_token st) {  
    stop_callback sc {st, [] {  
        cout << "stop requested";  
    }};  
} };
```

\*) if the stop has already been requested **prior** to the constructor's registration, the callable is executed in the thread constructing the `stop_callback`

# `std::stop_callback`

Which thread takes charge of the callback?

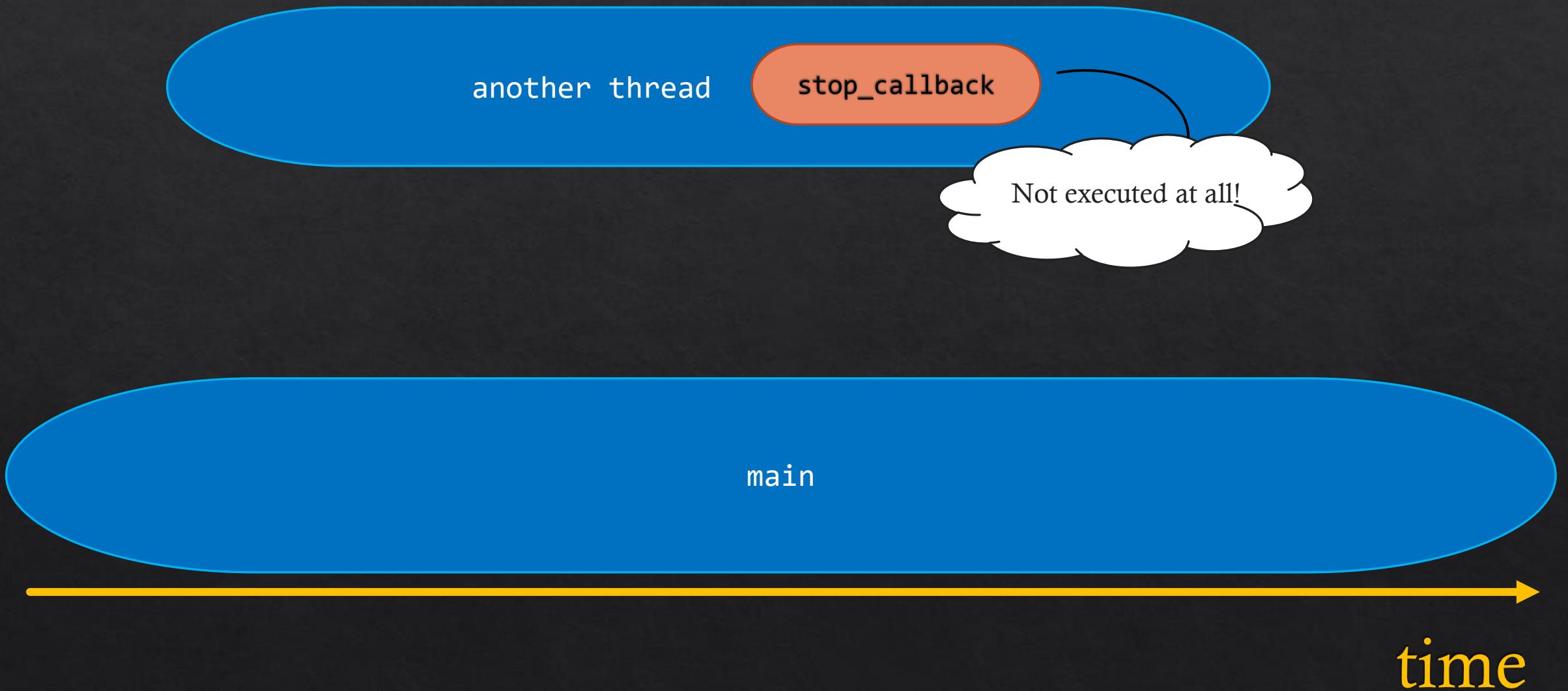
1. If the **thread finishes before** any cancellation requests, stop callbacks get automatically deregistered and dropped;
2. If the thread **has not constructed the callbacks yet** but a cancellation is requested, the callbacks get executed straight on the constructing thread;
3. Otherwise, the thread requesting the cancellation executes all the callbacks.



<https://wandbox.org/permlink/YdyHOZ63IXFnBKu5>

## SCENARIO 1

# std::stop\_callback



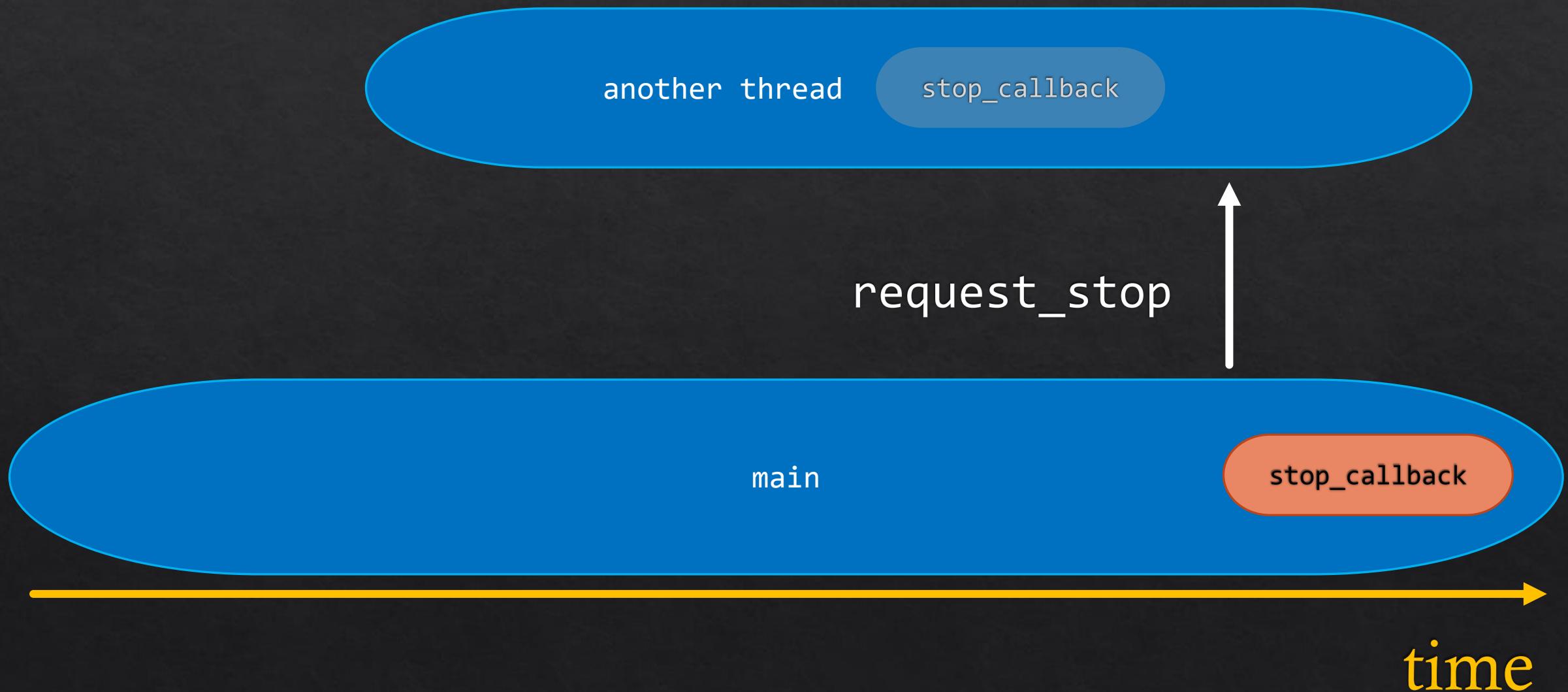
## SCENARIO 2

### std::stop\_callback



## SCENARIO 3

### std::stop\_callback



# Other concurrency features

- ❖ `std::latch` (single-use asynchronous waitable countdown)
- ❖ `std::barrier` (multi-use asynchronous waitable countdown)
- ❖ `std::counting_semaphore` (primitive to control access to a shared resource)
- ❖ Atomics improvements:
  - ❖ Low-level waiting
  - ❖ `shared_ptr` and `weak_ptr` specializations
  - ❖ `std::atomic_ref`
  - ❖ Full specializations for `float`, `double`, and `long double`



<https://wandbox.org/permlink/Uw2274dwdBeEjPkn>

# Library Features

# std::format

- ❖ We finally have fmtlib into the standard (some minor differences)!
- ❖ Format strings must be known at compile-time (e.g. string literals, constexpr char pointers)
  - ❖ C++23 will support compile-time checks of arguments
- ❖ For format strings that are computed at runtime, `std::vformat` is provided
- ❖ Alberto Barbati's talk at Italian C++ Conference 2020

```
auto luckInRome = std::format("Fortuna: {}", 23);
```



<https://wandbox.org/permlink/8VxVomYUhZlAufd7>

# std::span

- ❖ It's a reference (models `std::ranges::view`) to a **contiguous** sequence of elements
- ❖ Works with **contiguous\_iterator** only
- ❖ It's a **borrowed\_range** (usable as a temporary in range algorithms without dangling)
- ❖ Its type encodes if the number of elements is either **fixed** or **dynamic**
- ❖ In contrast to **string\_view**, it **can mutate** the referenced elements
- ❖ Multi-dimensional spans are not supported yet (`std::mdspan` will be supported in C++23)

```
int arr[3] = {1, 2, 3};  
  
std::span s = arr; // fixed-size automatically deduced
```



<https://wandbox.org/permlink/3hFwiEadmUp7nddS>

# std::source\_location

- ❖ `constexpr`-friendly class representing information about the source code, such as file names, line numbers, and function names

```
0 | cout << std::source_location::current().line(); // 0
1 | constexpr auto loc = std::source_location::current();
2 | cout << loc.line(); // 1
```



<https://wandbox.org/permlink/n5LFkgpIks1f6JwI>

# std::numbers

- ❖ We finally have mathematical constants (until now we have had macros only)!!!
- ❖ They are defined as inline constexpr variable templates

```
std::cout << std::numbers::pi << "\n"; // pi_v<double>  
std::cout << std::numbers::pi_v<float> << "\n";
```

# `std::midpoint` & `std::lerp`

- ❖ Not as straightforward as they could seem...
- ❖ Enjoy Fergus Cooper's talk [C++20: All the small things](#) (*C++ On Sea 2020*)
- ❖ You probably already know that  $(a + b)/2$  can lead to overflow
- ❖ So, the classical trick is just to use  $a + (b-a)/2$
- ❖ Nevertheless, things can go wrong when  $a$  is negative and  $(b-a)$  overflows...

# Bit manipulation functions

- ❖ We now have `<bit>` providing some common operations on raw bits  
(working on **unsigned integer types** only)

```
cout << popcount(12u); // number of ones in 12
cout << bit_width(12u); // number of bits needed for representing 12
```

- ❖ Maybe you are already using `bit_cast`  
(I have seen it called in several ways – e.g. `memcpy_cast`, `raw_cast`, `from_bits`)



<https://wandbox.org/permlink/HlIFcQEkHDCkP5wn>

# String utils

- ❖ We finally have member functions for checking if a `string` or `string_view` starts/ends with a certain prefix/suffix

```
vector<string> names { "charles", "anne", "edward", "matthew" };

auto who = ranges::find_if(names, [](const auto& s) {
    return s.starts_with("ed");
});

cout << "First entry starting with 'ed': " << *who << "\n";
```

# std::ssize

- ❖ Returns the size of a range converted to a **signed** type

```
for (auto i=0; i<ssize(rng); ++i)  
{  
    // ...  
}
```

# Container utils: contains

- ❖ Associative containers now provide the function `contains` to check if an element is present
- ❖ It's basically the same as `container.count(elem)>0`
- ❖ Supports **heterogenous lookup\***

```
set<string> attendees = { "marco", "antonio", "bigno", "andrea" };
if (attendees.contains("antonio")) {
    waiter.Order("Calzone", flags::Farcito);
}
```

\*) Since C++20, **unordered** containers support heterogeneous lookup

# Container utils: `erase_if`

- ❖ A non-member function `erase_if` has been introduced for the standard containers
- ❖ Basically, it's a self-contained **erase/remove idiom**
- ❖ Some containers have gained also a non-member `erase` function
- ❖ Returns the number of erased elements

```
vector<int> vals = {1, 2, 3, 4, 5};  
  
cout << erase_if(vals, [](auto i){ return i%2 == 0; }); // 2  
  
string s = "erase_underscores_from_here";  
  
cout << erase(s, '_'); // 3
```

# Container utils: `to_array`

- ❖ Creates a `std::array` from a **one dimensional** built-in array
- ❖ Supports also **move-only** types

```
auto arr1 = to_array("foo"); // array<char, 4>{'f', 'o', 'o', '\0'}
```

```
auto arr2 = to_array({1,2,3}); // array<int, 3>{1, 2, 3}
```

```
auto arr3 = to_array<long>({1,2,3}); // array<long, 3>{1, 2, 3}
```

# std::make\_\*\_for\_overwrite

- ❖ Same as `make_unique` and `make_shared` except that the object is **default-initialized**
- ❖ Overloads for array types are provided

```
// classical example  
auto i = make_unique_for_overwrite<int>();  
cin >> *i;
```

# Your new reaction to C++20



All the code snippets are also on GitHub:

<https://github.com/ilpropheta/cpp20-for-the-daily-job>



# Not covered today

- ❖ Concepts
- ❖ Ranges\*
- ❖ Coroutines
- ❖ Modules
- ❖ `std::chrono` additions (calendrical types and time zones support)
- ❖ `constexpreval`, `constexprexpr`, and most of the other compile-time computing additions
- ❖ Some other minor things like `std::string_view(begin, end)` constructor

\*) partially covered in June

# Some books to go further

- ❖ C++20 – The Complete Guide (Nicolai Josuttis)
- ❖ C++20: Get the Details (Rainer Grimm)
- ❖ Professional C++ - 5th Edition (Marc Gregoire)



span<Question>